A Framework for Handling Geophysical Datasets with the Matlab Programming Language

Frank Colberg

JPL/ CALTECH

MS 300-302, Pasadena 91109, CA

Version 1.0

June 2008

Contents

Abstract

This documentation serves as a user's guide, describing the framework of Matlab object oriented programming (OOP) features and the interface of Matlab generic ploting tool (MGPLOT) for analyzing and visualizing geophysical datasets. A to date unique approach aims to unify the way different datasets are handled. The framework encompasses different datatypes such as NetCDF, pure binary and ASCII, hence allowing for maximum flexibility. Typical datasets include the NCEP/ NCAR reanalyses, ROMS model output and ECCO2 products. Several hands-on examples are shown to demonstrate the flexibility and ease to use the software.

1 Introduction and Motivation

In todays scientific life a great amount of time is spend on analysing geophysical (i.e. Model, Observational) datasets. Within the last two decades or so the typical amount of data comprising a single dataset has increased drastically. This is partly the result of increasing computational power, which allows for finer (higher) resolution in Ocean/ Atmosphere Models to exist, but also due to more accurate satellite measurement, which lead to increasing amounts of produced data.

Particularly, model derived datasets are becoming extremely large and difficult to handle. So far, the Scientists are using programs based on software such as Matlab, Grads or Ferret or others. These software programs enable them to access and manipulate the data. Depending on the data processing tool, the degree of freedom or the flexibility in processing the data differs widely. Some data processing tools allow for easy and quick plotting while others allow for 'state of the art' data processing, but may be more difficult to handle. Nevertheless, Matlab is still the most powerful and most widely available tool to date.

However, although most or all geophysical datasets exhibit standard attributes such as coordinates, time indices, vertical levels etc., no integrated framework for the Matlab programming language exist that may enable the researcher to use any type of dataset in a similar or even the same fashion. Typically, dataset specific scripts exists that are adequately able to extract a certain type of data. A more advanced approach is to wrap these scripts around a function which allows for very specific events to happen. Nevertheless, documentation is sparse and most of the time the Scientists needs to implement their own data access routines and/ or to modify existing scripts to justify their needs. Most disturbingly, these methods differ for each and single dataset available.

Although many researches may prefer writing their own scripts just as to exert the maximum amount of control to how and why data is preprocessed, it is worth noting that this approach is still prone to unwanted errors and time consuming. Therefore, it is clear that this approach is not sustainable and that valuable time is repeatedly spend on tasks related to the assessment of variables of some dataset.

The presented framework intends to address some of these shortcomings. It makes use of Matlab's Object Orientated programming capabilities and effectively allows users to handle different datasets in the same fashion. In addition, a graphical user interface (MGPLOT) is naturally supported within this framework.

The framework has been developed specifically with the following thoughts in mind:

- same or similar handling of different datasets stored in different formats
- easy to use for Matlab experts and beginners
- easy and quick visualizations of datasets and indices
- sophisticated/state of the art/ dataset manipulations are supported due to command line integration
- a special focus is placed on the potential for expansion, allowing for the inclusion of individual dataset solutions,
- sustainable data centric programming approach via OOP

1.1 Design and Concept

As briefly stated in Section 1 this framework is based on the object orientated programming (OOP) features provided by Matlab. In the OOP world computer programs are viewed as a collection of individual units, or objects, that act on each other. This opposes the traditional view in which a program is seen as a collection of actions only. One of the advantages of OOP techniques over the linear programming techniques is that they enable programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from existing objects.

Hence, at the core of the OOP philosophy stands the idea of a program code that is easy to reuse and easy to expand. Arguably, the two most important aspects of OOP are:

a) Class *inheritance* and multiple *inheritance*. Child classes are more specialized versions of a class. They inherit attributes and behaviors from their parent classes, and can introduce their own. For example, the class Dog might have child-classes called Collie, Chihuahua, and GoldenRetriever. In this case, Lassie would be an instance of the Collie subclass. Suppose the Dog class defines a method called bark() and a property called furColor. Each of its child-classes (Collie, Chihuahua, and GoldenRetriever) will inherit these properties, meaning that the programmer only needs to write the code for them once. Each subclass can alter its inherited traits. For example, the Collie class might specify that the default furColor for a collie is brown-and-white. The Chihuahua subclass might specify that the bark() method produces a high pitch by default. Subclasses can also add new members.

In fact, inheritance is an 'is-a' relationship: Lassie is a Collie. A Collie is a Dog. Thus, Lassie inherits the methods of both Collies and Dogs. (Source: WIKIPEDIA).

b) *Polymorphism* in OOP is the ability of objects belonging to different data types to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior. One method, or an operator such as +, -, or *, can be abstractly applied in many different situations. If a Dog is commanded to speak(), this may elicit a bark(). However, if a Pig is commanded to speak(), this may elicit an oink(). They both inherit speak() from Animal, but their derived class methods override the methods of the parent class. This is Overriding Polymorphism. (Source: WIKIPEDIA).

Matlab naturally uses these two OOP features. For example, adding integer values in Matlab calls a different *plus.m* function than does adding two matrices. It is easy to grasp how one might expand this approach towards geophysical datasets, with different datasets needing different strategies for data extraction.

In line with OOP philosophy this framework defines a base class (ModelData) which is capable of handling standard reading/ writing tasks. In addition, this class implements the infrastructure that all other child classes may use. This includes elementary functions allowing to set/ get object attributes, but also more complex tasks like adding additional geophysical fields which may exist due to the specific dataset structure. The base class does not know about the specific geophysical properties of a certain dataset and hence only the basic (native) variables are initialized when using the ModelData object.

This concept is extended by defining child classes that contain the specific information about a certain dataset (such as names of variables, coordinates, z-levels, etc.). More specifically this means assigning each substantial different dataset its own class or object. This is similar to what has been described for the dog-class example above.

For example a ROMS output file would be assigned a *roms* object. An ECCO output file would be assigned an *ecco* object and so forth. Each of these objects are child classes of the ModelData class. Hence they inherent some basic e.g. netcdf reading capabilities. Furthermore each class implements a method *getdata* which knows exactly how to extract and access the data for the specific data type. In accordance with *polymorphism* this method may or may not overwrite the original *getdata* method as implemented by ModelData, but may use ModelData capabilities to perform basic reading tasks.

2 ModelData in Depth

If a certain datafile is assigned its class the returned object instance should be able to provide the following informations and tasks: (a) know what variables are stored in the datafile, (b) know their physical meaning, (c) know what other physical variables can be derived from them, and (d) provide a way to get the data. This framework achieves this by defining identities.

2.1 Framework identities

The framework distinguishes between three fundamental identities. The **natives**, **names** and **derived** identities. The **natives** identitiy denotes all variables that are stored in the datafile associated with the object-instance. The **derived** identity results from the combination of two or more native identities or from the ability to *derive* a different variable from a *native* one. For example a dataset may consist of u, v velocities, which in this case are the native variables of the dataset. Taken together they form the absolute velocity and the velocity vector field. These are **derived** variables. Another example of a derived variable may be the zonal/ meridional temperature transport, which relates to T and u, v.

The **names** identity translates the datafile specific variable names to real world **names** or names that the framework knows. Within the framework the names identity is unique for each existing geophysical variable. That means sea surface temperatures e.g. will always be denoted as *Sea Surface Temperature* in the **names** context. The framework checks at initialization time for these translations and then dynamically creates the possible valid relationships associated with each datafile.

Therefore, the **names** identity allows for the implementation of routines that tie together the information of one or more native variables and decide if another variable may be derived from it/ them.

2.2 Adding Objects

The OOP approach allows to override basic functions as described by *polymorphism* (Section 1.1). This concept is used in this framework to allow for the possibility to add two object instances of the same class. The resulting object variable contains the combined information of the two individual instances $\underline{\text{plus}}$ any additional information that is only available as a consequence of the combination of the two individual object instances.

This approach has some practical consequence. For example consider the situation in which the u velocity of an arbitrary data/ model output is stored in one file and the v velocity in another file. None of two files alone is able to provide the necessary information to plot the velocity vector or the absolute velocity. In practice one needs to extract the data from both files. When using this framework it is possible to extract the absolute velocity by adding the two object instances. This may seem trivial, but provides a powerful way to deal with plotting datasets and is extensively used within MGPLOT (see Section 3). For examples see Section B.

2.3 Viewing Variables

The framework implements Matlab typical syntax *get* to view and 'browse' through the content of an object instance. The above discussed identities *natives, names, derived* are used to decide what kind of information is to be displayed. For specific syntax usage see Section A, and the examples Section B

2.4 Extracting the data

The way to extract the actual data is similar to the browsing process. The framework implements a function *getdata* which returns variable output arguments. Each data class share the same basic functionality but may extend it to fulfill specific dataset requirements.

For example, the ECCO2 model data output is known to be on 6 faces, so it is desirable to being able to extract only one or all of them. Hence, the ecco2 class may implement something like *getdata(face 1)*etc.. A roms class on the other hand does not need to know about faces but rather about σ-coordinates and how to interpolate them onto z-levels. For specific syntax usage see Section A, and the examples Section B.

2.5 Customize Object Instances

One strength of OOP is to set specific attributes for object instances. Allowing each instance to be unique but still share some comon attributes given by the parent class. This means that for example each setup of a certain model type is automatically supported if the basic model features are known. This is useful when dealing for example with the regional modelling system (ROMS) which can be setup quickly in most regions of the world ocean. The framework, however, only needs one *roms* class. Each instance may be related to a different model setup. In general the user does not need to change or set individual model attributes. However, sometimes it may be inevitable. Then the typical Matlab syntax *set* can be used to override default variable definitions. For specific syntax usage see Section A, and the examples Section B.

2.6 Saving Object Instances

It is possible to save object instance the same way as it is to save any other variable in Matlab. Saving object instances may be useful if a great amount of individual instances have been added to form one object.

3 The Matlab Generic Plotting Tool (MGPLOT)

MGPLOT is part of a framework described in the previous Sections (1-2) and is aiming for easy plotting sensations with only little knowledge of the Matlab processing language. It naturally supports object classes if they are part of the above described framework. At the current development stage the following datasets are supported by the framework and hence MGPLOT:

- NCEP monthly means
- ROMS (Ruttgers, UCLA, IRD)
- ECCO2
- WOA

Figure 6 shows a snapshot of MGPLOT. Please refer to Figure 6 and its captions that provide the basic information of how to manage the GUI.

4 Setup

In order to setup the framework add the path of the installation directory either at Matlab runtime or in the Matlab *startup.m* file. i.e. *addpath('thepath')* Once the path is set *MGPLOT* (the graphical plotting tool) can be started by typing: *mgplot* in the command line.

During the initialization process MGPLOT adds a number of additional pathes to the environment. Without adding these pathes MGPLOT does not function. The reason why this task is hidden from the user is to make the installation as easy as possible. However, in some cases these pathes may interfere with the general users settings/ environment. In this case they can be removed by typing: *mgplot_rmpath* on the command line.

In case the user wishes to benefit from the data classes and the plotting utilities only, the additional pathes must be set manually by typing: *mgplot_addpath* on the command line. Again this will add all the pathes.

In order to use this framework NetCDF reading capabilities are required and need to be installed for Matlab. Note, the setup has been tested under Linux only. Problems with path names and/ or slash directions may occur when using Matlab under Windows/ Mac.

MGPLOT lets the user customize the default pathes for different datasets. This is done in file: *mgplot_initialize.m*. For each data class that is supported in MGPLOT a path variable needs to be declared. The path could just be empty (default).

The setup procedure for the framework and MGPLOT:

- Add the path to the MGPLOT directory in the Matlab *startup.m* file or at runtime
- Start Matlab and type *mgplot* in the command line to start the GUI
- Edit *mgplot_initialize.m* to customize the default data pathes used when creating a new object with MGPLOT (optional)
- Start Matlab and type *mgplot_addpath* to use the data-class definitions and other tools without starting the GUI
- Type *mgplot_rmpath* to remove the pathes associated with MGPLOT from the Matlab environment

A Classes/ Declaration/ Usage

This Section lists the to date implemented class definitions. The framework has been developed using Matlab version 7.4.0.287 (R2007a). The latest Matlab version implements a new more timely approach for defining classes. The new class definition may be used in future versions of this tool.

A.1 Class ModelData

ModelData is the base class from which all other data-classes are derived. The basic task of ModelData is to provide NetCDF reading capabilities and to implement the general structure for storing dataset specific variable-information. The ModelData class implements the **natives** identity and has methods that allow for the creation of the **names** and **derived** identity. These methods are only accessible from the child classes. ModelData adds implements a fourth identity: the **other** identity. This identity allows for viewing and retrieving variables with less than two dimensions, such as coordinates and depth matrices.

In addition, the ModelData class implements a climatology and an anomaly method when using *getdata*. The keywords are *climatol* and *anomaly* respectively. These two methods may be invoked if the time index of a NetCDF-file is greater than one. Additional features associated with the *get/ set* and *getdata* command are listed below.

Create object instance:
OBJ = *ModelData* (**filename**)

Public methods implemented by ModelData:

- *get* ⇒ function: Provides access to object attributes
- *set* ⇒ function: Allows to set object attributes
- *getdata* ⇒ function: Provides access to object data
- *plus* operator ⇒ function: Adds two objects

Private methods implemented by ModelData:

- *setfieldforall* ⇒ function: Sets field attributes for all variables apparent in an object. This function can be invoked via the set function. Keyword is *forall*
- *addnativefield* ⇒ function: Adds a native variable to the object
- *addchildfield* ⇒ function: Adds a child variable to the object
- *addchildderivatives* ⇒ function: Adds possible derived variables to the object. This function can be invokes by a child class only and needs additional information.

- *getdata_after_extract_data* ⇒ function: Adds some functionality to getdata methods (i.e. permute, reshape, anomaly, climatology).
- *getdata_after_extract_sum* ⇒ function: Adds functionality to getdata methods
- *getdata_after_extract_average* ⇒ function: Adds functionality to getdata methods

Access object attribute:
VALUE = *get* (OBJ, **identity**)
with **identity**:

- 'natives' ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- 'other' ⇒ VARIABLE
- 'fields' ⇒ data
- 'fields' ⇒ other

with ATTRIBUTE for each VARIABLE

- 'varis' *string* of type char ⇒ denotes the identitity
- 'var' *string* of type char wrapped in *cell* ⇒ name of variable
- 'dimension' *scalar* of type integer ⇒ dimension
- 'levels' *scalar* of type integer ⇒ number of vertical levels
- 'fname' *string* of type char ⇒ path to object file name
- 'size' *vector* of type integer ⇒ size of VARIABLE
- 'tndx' *scalar* of type integer ⇒ number of time records
- 'ID' ⇒ not used at present
- 'gname' *string* of type char ⇒ path to grid file if given
- 'precision' *string* of type char ⇒
- 'netcdf' ⇒ not used at present
- 'i' *scalar* of type integer ⇒ number of i-curves
- 'j' *scalar* of type integer ⇒ number of j-curves

- 'region' *vector* of type integer with *vector=[1 i 1 j]* ⇒ denotes the region of the variable
- 'levelsfull' *vector* of type integer wrapped in *cell* ⇒ needed for MGPLOT
- 'tndxfull' *vector* of type integer wrapped in *cell* ⇒ needed for MGPLOT
- 'levelsdepth' ⇒ depth matrix

Set object attribute:
OBJ = *set* (OBJ, **identity**)
with **identity**:

- 'natives ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- 'other' ⇒ VARIABLE
- 'fields' ⇒ data
- 'fields' ⇒ other

with ATTRIBUE same as for get.
Extract data:
VAR = *getdata* (OBJ, 'natives', **VARNAME**, *options*)
with *options*:

- 'region', *vector* of type integer with *vector* = $[i_1\, i_2\, j_1\, j_2]$ ⇒ returns the specified region for the selected variable
- 'tndx', *vector* of type integer ⇒ returns the specified time index (indices) for the selected variable
- 'levels', *vector* of type integer ⇒ returns the specified vertical level(s) for the selected variable
- 'npoints', *scalar* of type integer ⇒ extracts every n_{th} points of the variable
- 'anomaly', *vector* of type integer *[1..max(numberoftimerecords) orderofanomay]* ⇒ calculates the anomaly of given order (last vector value), with a mean calculated over *vector[1..end(-1)]*
- 'climatol', *vector* of type integer ⇒ returns the climatology of given order

A.2 Class ecco2

The ecco2-class implements reading capabilities for native (i.e. binary) ecco2 output. Hence, it mainly uses the structure for storing datatype specific information given by ModelData. It overrides the ModelData *getdata* method. Most convienently the class allows to extract individual ecco2-faces but will also interpolate ecco2-data onto a regular $0.25deg$ grid. This behaviour is controlled with the $face$ parameter in *getdata*. Furthermore, the ecco2-class allows to specify the latitude and longitude for the extracted region. The keyword is *lonlat* in the *getdata* method. This functionality is only available when global interpolation is chosen. In addition it is possible to loop over different ecco2 files that are in the same path as the initialization file via setting the *sequence* parameter in *getdata*. The parameter can either be a *vector* with integers denoting the ecco2 files or a *string* with value *allnames*. In the latter case the loop will be over all the files for the selected variable that are in the original file path.

The ecco2-class sets the time for the datafile in the attribute *realtime*. Whenever the user changes either the *timesteplength* or the *timestep*, the ecco2-class will update *realtime*. Internally, the function ts2dte.m is used for that purpose. The ecco2-class relies heavily on external helper files that provide the relevant grid information including the angles that are needed for the interpolation onto the regular grid. This information is stored in the attributes: *xcdata*, *ycdata*, *anglecssn*. The class assumes that the *precision* for a variable is allways $real * 4$ except for the temperature where it is $real * 8$. However, the precision information for any variable can be changed using the *set* command.

Create instance:
OBJ = *ecco2* (**filename**)
Private methods implemented by ecco2:

- settime.m
- getcoordinates.m
- setlevelsforall.m

External ecco2 functions: readbin.m, ts2dte.m, rdda.m

Get object attribute:
VALUE = *get* (OBJ, **identity**)
with **identity**

- ’natives’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- ’derived’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- ’names’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE

with ATTRIBUTE for each VARIABLE:

- ’timestep’ ⇒ ECCO2 timestep
- ’timesteplength’ ⇒ ECCO2 time step length.
- ’realtime’⇒ realtime
- ’xcdata’ ⇒ path to XCDATA.dat
- ’xcdata’ ⇒ path to XCDATA.dat
- ’precision’ ⇒ precision for VARIABLE
- ’anglecssn’ ⇒ path to Angle_CSSN .mat

Set object attribute:
OBJ = *set* (OBJ, **identity**)
with **identity**

- ’natives’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- ’derived’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE
- ’names’ ⇒ VARIABLE ⇒ ATTRIBUTE ⇒ VALUE

with ATTRIBUTE for each VARIABLE:

- ’timestep’, *scalar* of type integer ⇒ set ECCO2 timestep. When set recalculates the attribute ’realtime’
- ’timesteplength’ *scalar* of type integer ⇒ set ECCO2 time step length. When set recalculates the ’realtime’
- ’xcdata’, *string* of type char ⇒ set path to XCDATA.dat
- ’xcdata’, *string* of type char ⇒ set path to XCDATA.dat
- ’precision’,*string* of type char with value *real*4* or *real*8* ⇒ set precision for VARIABLE

- 'anglecssn', *string* of type char $\Rightarrow$ set path to Angle_CSSN.mat

Extract data:
variableoutput = *getdata* (OBJ, **identity**, *options*)
with **identity**

- 'natives' $\Rightarrow$ VARIABLE
- 'derived' $\Rightarrow$ VARIABLE

with **variableoutput**:

1. [lon, lat] if *options(1)* $\Rightarrow$ 'coordinates', *options(2)* $\Rightarrow$ 'horizontal'
2. [z-matrix] if *options(1)* $\Rightarrow$ 'coordinates', *options(2)* $\Rightarrow$ 'vertical'
3. [var] for any given *options*
4. [var, lon, lat] for any given *options* except for 1. and 2.
5. [var, lon, lat, mask] for any given *options* except for 1. and 2.
6. [var, lon, lat, mask, var1, var2] for any given *options* except for 1. and 2.

with *options*

- 'lonlat', *vector* containing [longitude1 longitude2 latitude1 latitude2]
- 'face' *vector* of type integer either *0* or *[1..max(numberoffaces)]*
  - if *0* returns global interpolated grid with size 1440×720
  - otherwise returns face(s) given in *vector*
- 'sequence'
  - *vector* of type integer denoting the specific ecco2 files or
  - *string* of type char with value *allnames*

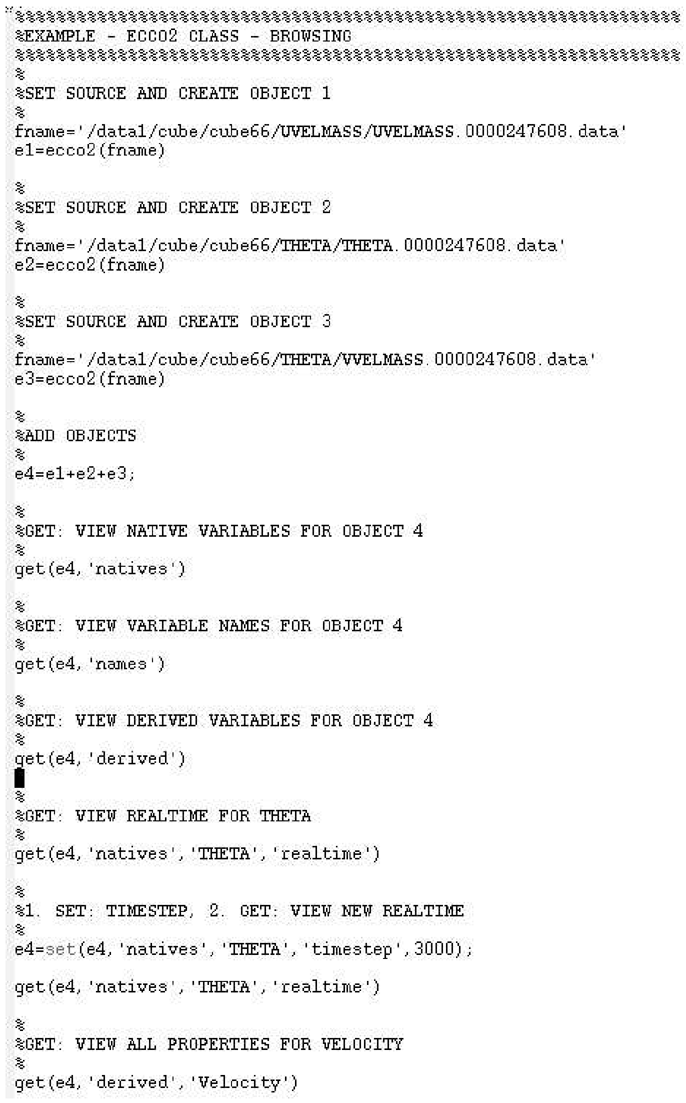
References

B Examples

The following datasets/ models are covered by the examples section:

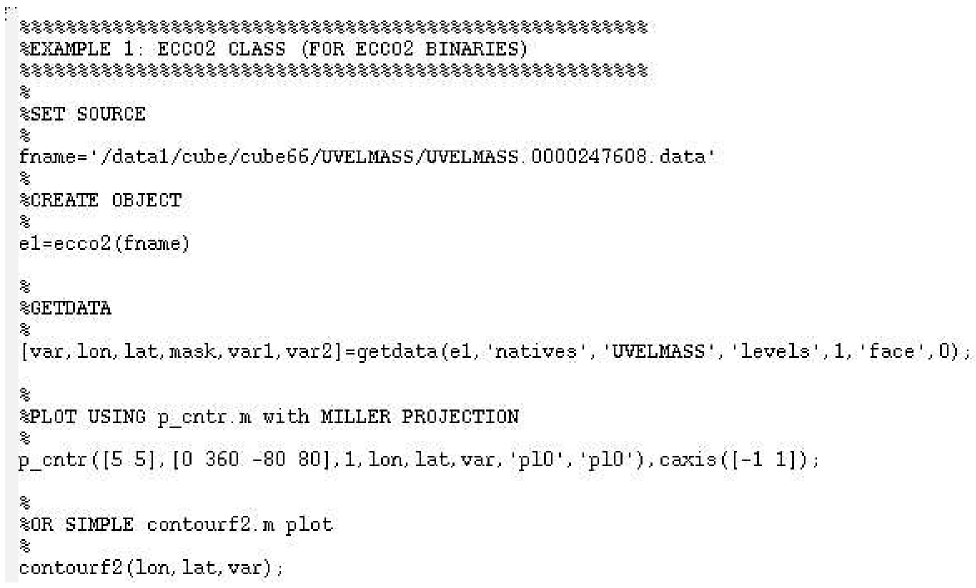
- ECCO2
- ROMS
- NCEP/NCAR Reanalysis Monthly Means
  (http://www.cdc.noaa.gov/cdc/data.ncep.reanalysis.derived.surfaceflux.html)

Note, to successfully reproduce the examples the framework needs to be implemented correctly. Please refer to Section 4 for the general setup procedure.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EXAMPLE - ECCO2 CLASS - BROWSING
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%SET SOURCE AND CREATE OBJECT 1
%
fname='/data1/cube/cube66/UVELMASS/UVELMASS.0000247608.data'
e1=ecco2(fname)

%
%SET SOURCE AND CREATE OBJECT 2
%
fname='/data1/cube/cube66/THETA/THETA.0000247608.data'
e2=ecco2(fname)

%
%SET SOURCE AND CREATE OBJECT 3
%
fname='/data1/cube/cube66/THETA/VVELMASS.0000247608.data'
e3=ecco2(fname)

%
%ADD OBJECTS
%
e4=e1+e2+e3;

%
%GET: VIEW NATIVE VARIABLES FOR OBJECT 4
%
get(e4,'natives')

%
%GET: VIEW VARIABLE NAMES FOR OBJECT 4
%
get(e4,'names')

%
%GET: VIEW DERIVED VARIABLES FOR OBJECT 4
%
get(e4,'derived')

%
%GET: VIEW REALTIME FOR THETA
%
get(e4,'natives','THETA','realtime')

%
%1. SET: TIMESTEP, 2. GET: VIEW NEW REALTIME
%
e4=set(e4,'natives','THETA','timestep',3000);

get(e4,'natives','THETA','realtime')

%
%GET: VIEW ALL PROPERTIES FOR VELOCITY
%
get(e4,'derived','Velocity')
```

Figure 1: Example: Viewing the content of an object-instance.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EXAMPLE 1: ECCO2 CLASS (FOR ECCO2 BINARIES)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%SET SOURCE
%
fname='/data1/cube/cube66/UVELMASS/UVELMASS.0000247608.data'
%
%CREATE OBJECT
%
e1=ecco2(fname)

%
%GETDATA
%
[var,lon,lat,mask,var1,var2]=getdata(e1,'natives','UVELMASS','levels',1,'face',0);

%
%PLOT USING p_cntr.m with MILLER PROJECTION
%
p_cntr([5 5],[0 360 -80 80],1,lon,lat,var,'pl0','pl0'),caxis([-1 1]);

%
%OR SIMPLE contourf2.m plot
%
contourf2(lon,lat,var);
```

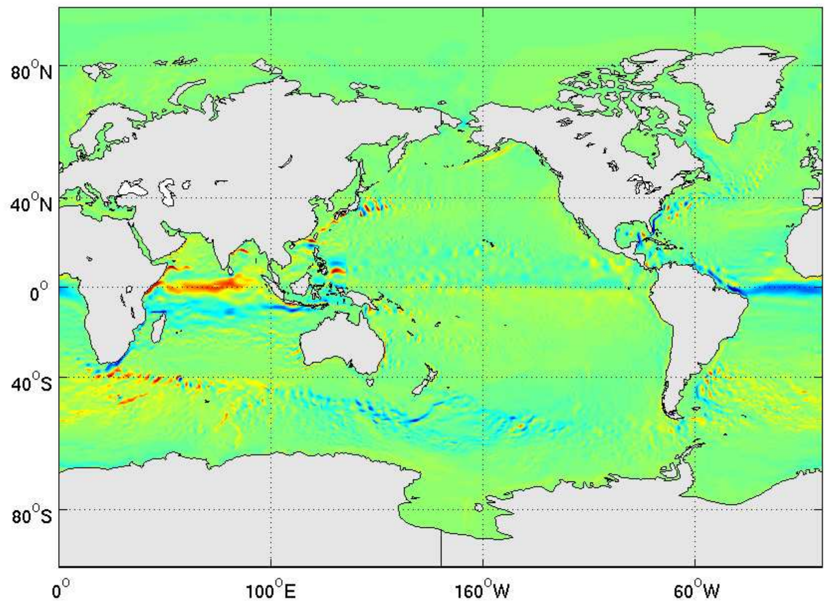
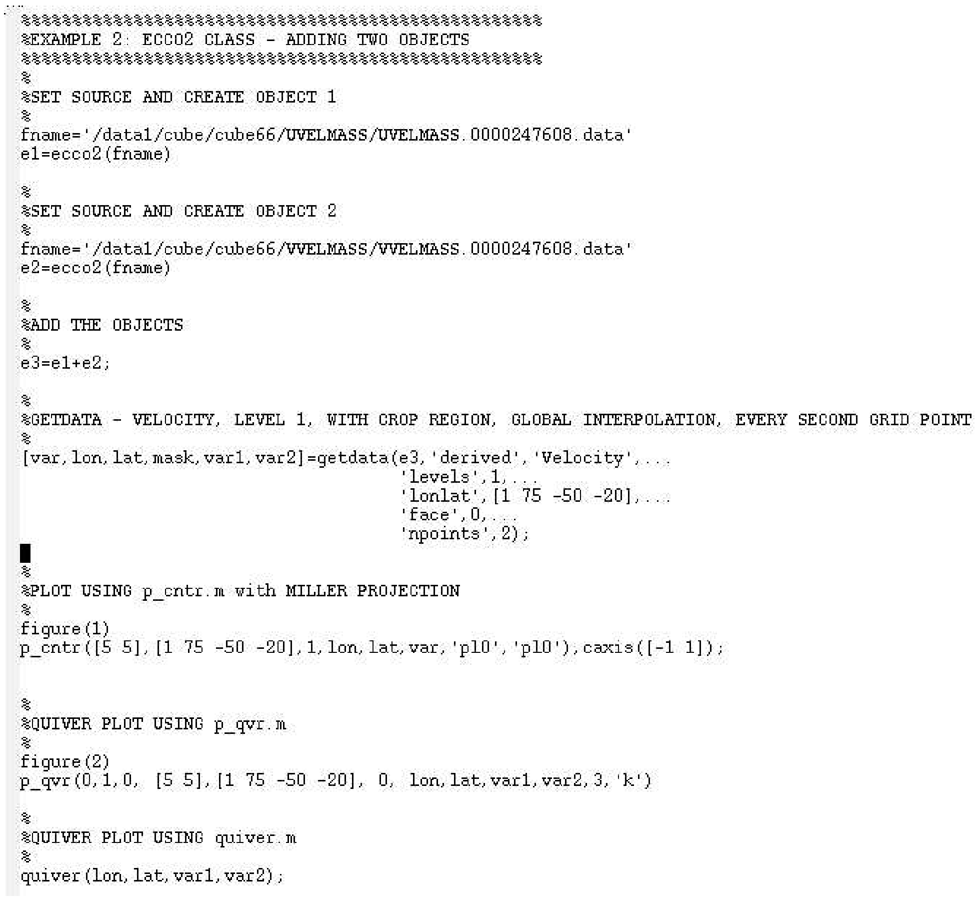


Figure 2: Example: Extract and plot the content of an object-instance.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EXAMPLE 2: ECCO2 CLASS - ADDING TWO OBJECTS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%SET SOURCE AND CREATE OBJECT 1
%
fname='/data1/cube/cube66/UVELMASS/UVELMASS.0000247608.data'
e1=ecco2(fname)

%
%SET SOURCE AND CREATE OBJECT 2
%
fname='/data1/cube/cube66/VVELMASS/VVELMASS.0000247608.data'
e2=ecco2(fname)

%
%ADD THE OBJECTS
%
e3=e1+e2;

%
%GETDATA - VELOCITY, LEVEL 1, WITH CROP REGION, GLOBAL INTERPOLATION, EVERY SECOND GRID POINT
%
[var,lon,lat,mask,var1,var2]=getdata(e3,'derived','Velocity',...
                                      'levels',1,...
                                      'lonlat',[1 75 -50 -20],...
                                      'face',0,...
                                      'npoints',2);

%
%PLOT USING p_cntr.m with MILLER PROJECTION
%
figure(1)
p_cntr([5 5],[1 75 -50 -20],1,lon,lat,var,'pl0','pl0'),caxis([-1 1]);


%
%QUIVER PLOT USING p_qvr.m
%
figure(2)
p_qvr(0,1,0,  [5 5],[1 75 -50 -20],  0,  lon,lat,var1,var2,3,'k')

%
%QUIVER PLOT USING quiver.m
%
quiver(lon,lat,var1,var2);
```

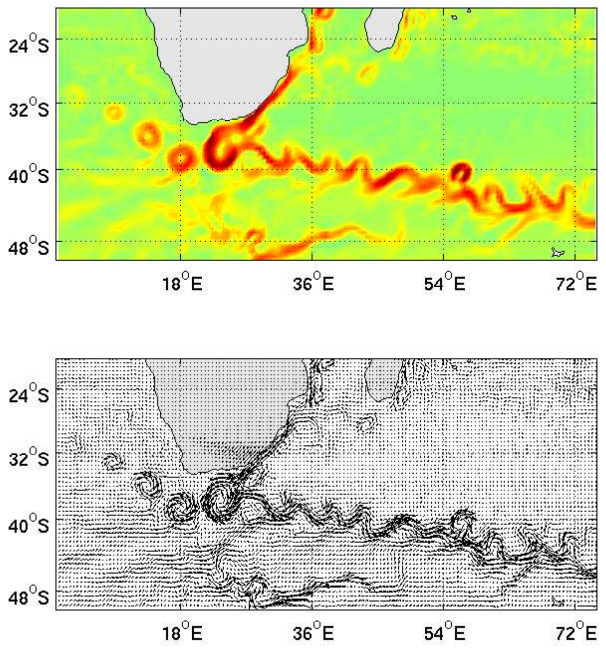
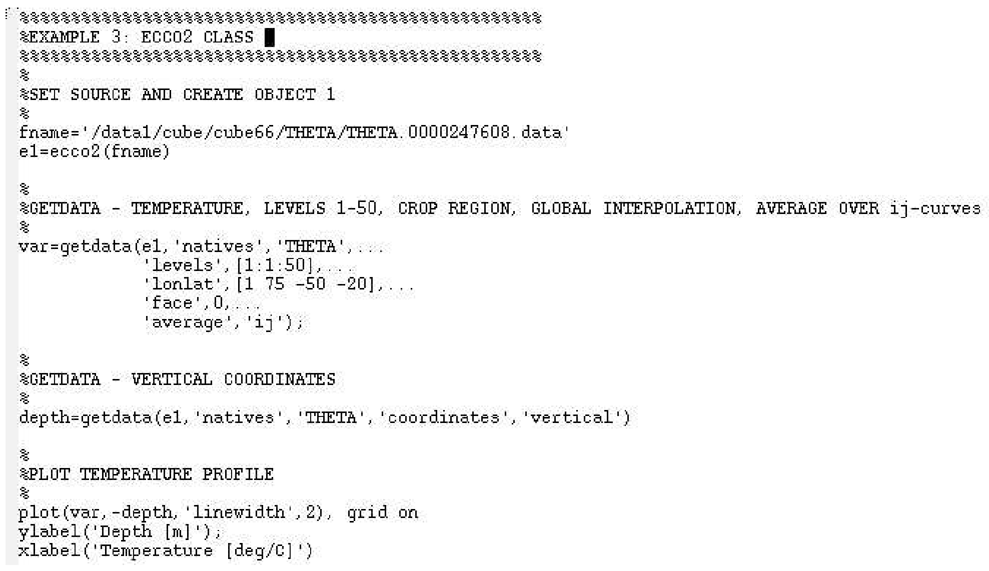


Figure 3: Example: Extract and plot the content of an object-instance.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EXAMPLE 3: ECCO2 CLASS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%SET SOURCE AND CREATE OBJECT 1
%
fname='/data1/cube/cube66/THETA/THETA.0000247608.data'
e1=ecco2(fname)

%
%GETDATA - TEMPERATURE, LEVELS 1-50, CROP REGION, GLOBAL INTERPOLATION, AVERAGE OVER ij-curves
%
var=getdata(e1,'natives','THETA',...
            'levels',[1:1:50],...
            'lonlat',[1 75 -50 -20],...
            'face',0,...
            'average','ij');

%
%GETDATA - VERTICAL COORDINATES
%
depth=getdata(e1,'natives','THETA','coordinates','vertical')

%
%PLOT TEMPERATURE PROFILE
%
plot(var,-depth,'linewidth',2), grid on
ylabel('Depth [m]');
xlabel('Temperature [deg/C]')
```

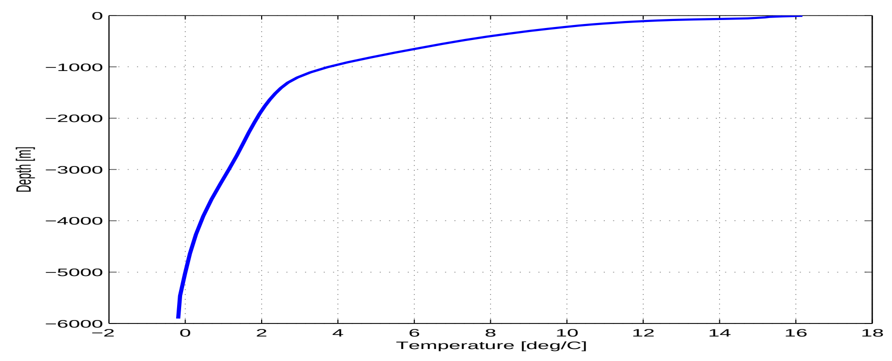
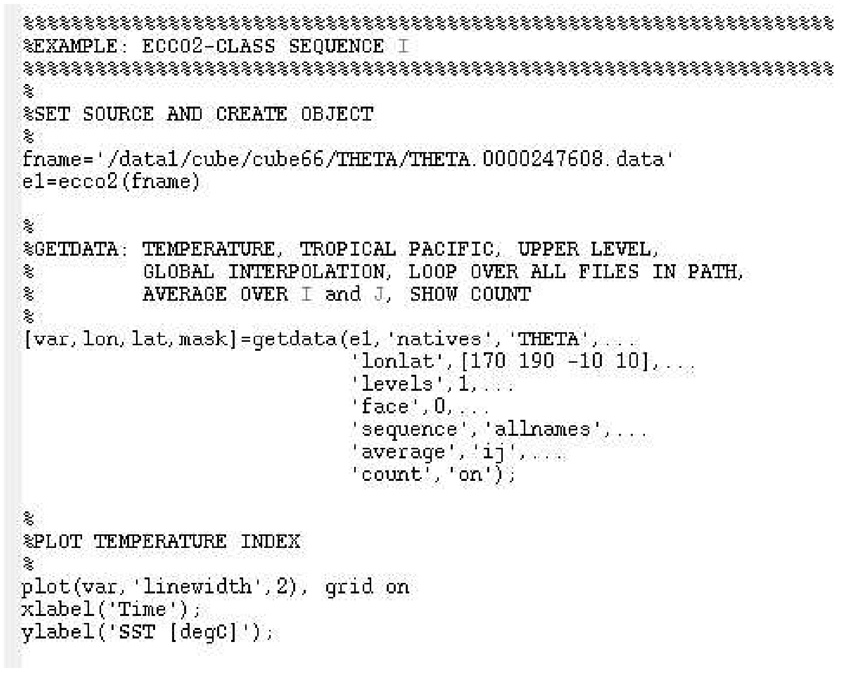


Figure 4: Example: Extract and plot the content of an object-instance.



```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%EXAMPLE: ECCO2-CLASS SEQUENCE I
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%SET SOURCE AND CREATE OBJECT
%
fname='/data1/cube/cube66/THETA/THETA.0000247608.data'
e1=ecco2(fname)

%
%GETDATA: TEMPERATURE, TROPICAL PACIFIC, UPPER LEVEL,
%         GLOBAL INTERPOLATION, LOOP OVER ALL FILES IN PATH,
%         AVERAGE OVER I and J, SHOW COUNT
%
[var,lon,lat,mask]=getdata(e1,'natives','THETA',...
                            'lonlat',[170 190 -10 10],...
                            'levels',1,...
                            'face',0,...
                            'sequence','allnames',...
                            'average','ij',...
                            'count','on');

%
%PLOT TEMPERATURE INDEX
%
plot(var,'linewidth',2), grid on
xlabel('Time');
ylabel('SST [degC]');
```

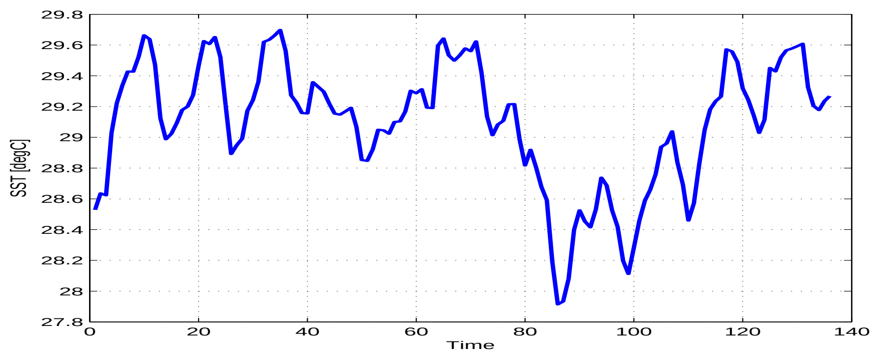
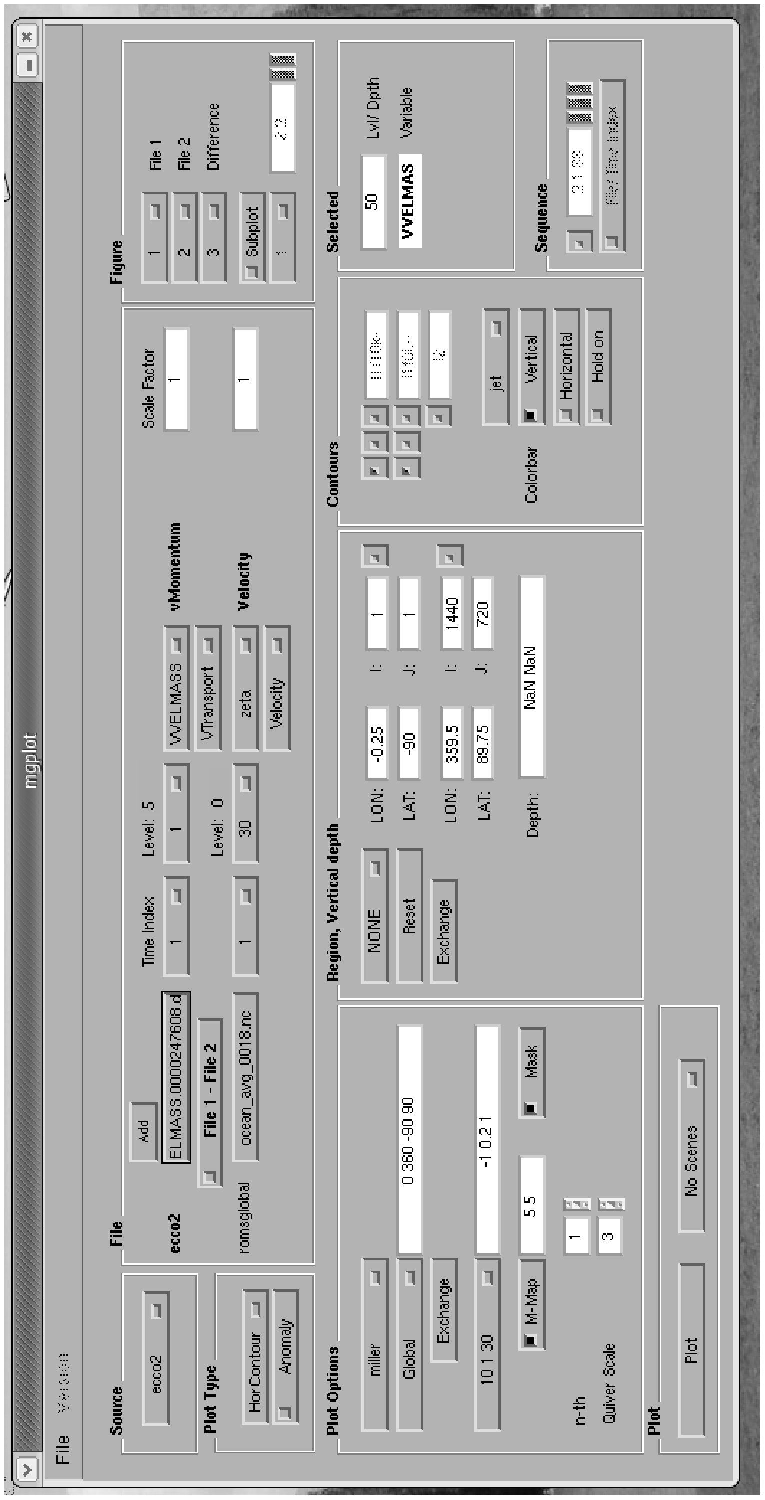


Figure 5: Example: Extract and plot the content of an object-instance. This examples does only work for MGPLOT-1.4-ecco2 version !!



Source: ecco2

Plot Type: Hor Contour; Anomaly

ecco2: VELMASS.0000247608.d; Time Index: 1; Level: 5; 1; VVELMASS; vMomentum; VTransport; Scale Factor: 1

romsglobal: ocean_avg_0018.nc; Time Index: 1; Level: 0; 30; zeta; Velocity; Velocity; Scale Factor: 1

Figure: 1 File 1; 2 File 2; 3 Difference; Subplot

Plot Options: miller; Global 0 360 -90 90; 10 1 30 -1 0.2 1; M-Map 5 5; Mask; n-th 1; Quiver Scale 3

Region, Vertical depth: NONE; LON: -0.25 I: 1; LAT: -90 J: 1; LON: 359.5 I: 1440; LAT: 89.75 J: 720; Depth: NaN NaN

Contours: jet; Colorbar: Vertical; Horizontal; Hold on

Selected: 50 Lvl/ Dpth; VVELMAS Variable

Figure 6: Snapshot of the Matlab Generic Plotting Tool (MGPLOT).

1 Source Panel: Choose base object (needs to be selected prior opening file !!)

2 File Panel: (a, b) Open File 1/2, (c) Add object instance to file, (d) Select difference plot between File 1/2, (e, f) Select time index for File 1/2, (g, h) Select level for File 1/2, (i, j) Select variable for File 1/2, (k, l) Select derived variable for File 1/2, (m, n) Choose scale factor for variable of File 1/2

3 Figure Panel: (a, b) Select figure for File 1/2, (c) Figure for difference plot, (d) Select subplot

4 Plot Type Panel: (a) Choose plot type, (b) Select anomaly plot (not yet implemented)

5 Plot Options Panel: (a) Select projection, (b, c) Select specific area, (d) Try matching selected area with grid points of selected object (does not allways work, depends on algorithm convergence), (e, f) Select caxis, (g, h) Select M-Mapping toolbox (on/ off, default is on, otherwise projections do not work), (i) Select mask (on/ off) not implemented yet, (j) Select to plot every n_{th} point, (k) Select scale factor for quiver plot

6 Region, Vertial depth Panel: (a) Selected crop regions, (b) Reset region to original size, (c) Exchange region with M-Mapping area, (d) Select region to crop, (e, f) Press and choose point on map (works only if resolution is not too high - need new algorithm), (d) Select depth for vertical or transport plot (NaN NaN means plot over the whole colum, otherwise: -500 0)

7 Contours Panel: Upper: Choose kind of contour plot for positive values, with (a) pcolor, (b) contourf, (c) pcolor with contour line-style specified in (d), Middle: For negative values, Lower: For zero contour line, (d) Change contour style and labelling (see help contourf2 for details) (e) Select colormap (f, g) Select colorbar (i) Hold on button

8 Selected Panel: (a) Display level to plot for variable: Select negative value for σ-coordinate interpolation, (b) Shows selected variable to plot

9 Sequence Panel: (a) Select sequence, (b) Choose if sequence over time index or over File (only supported for ROMS model output at the moment)

10 Plot Panel: (a) Plot variable, (b) Select scene: Calls specific routine which implements own plotting script